A Hoare Calculus for the Verification of Synchronous Languages

Manuel Gesell, Klaus Schneider
http://es.cs.uni-kl.de

Embedded Systems Group
University of Kaiserslautern

International Open Workshop on Synchronous Programming
2011
1. Introduction

2. Preliminaries
   - Quartz
   - Hoare Calculus

3. A Hoare Calculus for Quartz

4. Conclusion
Outline

1. Introduction

2. Preliminaries
   - Quartz
   - Hoare Calculus

3. A Hoare Calculus for Quartz

4. Conclusion
What is this Talk about?

- synchronous languages
- Quartz
- Hoare calculus
- synchronous tuple assignment form
- a Hoare calculus for Quartz
Synchronous Model of Computation

- abstract time to sequence of reactions (instants)
- each variable has one value per instant
- inputs and outputs are read and produced for an instant
- coincides with clock-cycles of synchronous circuits
  - gate delays mimic computation
  - one value per wire for each clock cycle
- instants are a logical time-scale

Synchronous Trace

\[ R_0 \quad \quad R_1 \quad \quad R_2 \quad \quad R_3 \quad \quad \ldots \]
Synchronous Languages

- implement the synchronous model
- can be used for hardware and software
- data-flow oriented languages
  - Lustre
  - Signal
- control-flow oriented languages (imperative)
  - Quartz
    - developed in our working group
    - Averest toolset
  - Esterel
Outline

1. Introduction

2. Preliminaries
   - Quartz
   - Hoare Calculus

3. A Hoare Calculus for Quartz

4. Conclusion
# Outline

1. Introduction
2. Preliminaries
   - Quartz
   - Hoare Calculus
3. A Hoare Calculus for Quartz
4. Conclusion
Quartz Example

```module P1 (nat ?i1,?i2,o1,o2)
{
    nat x;
    loop {
        o1 = i1 + i2;
        x = i1;
        pause;
        o1 = o2 + i1 + x;
        o2 = i2;
        x = 2;
        pause;
        if (i1 > 4)
            o1 = i1;
        o2 = i1 + o1;
        pause;
    }
}
```

- **pause** marks end of a step
- i1, i2 are inputs, o1, o2 are outputs, x is a local variable

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>i1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>i2</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>x</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>o1</td>
<td>3</td>
<td>8</td>
<td>8</td>
<td>12</td>
<td>7</td>
</tr>
<tr>
<td>o2</td>
<td>0</td>
<td>4</td>
<td>11</td>
<td>11</td>
<td>0</td>
</tr>
</tbody>
</table>
Quartz Statements

- assignments: \( x = \alpha, \text{next}(x) = \alpha \)
- end of step: \( \text{pause} \)
- conditional execution: \( \text{if}(\gamma) \ldots \text{else} \ldots \)
- loops: \( \text{while}(\gamma) \{ \ldots \}, \text{loop} \{ \ldots \} \)
- abortion: \( \text{abort} \ldots \text{when}(\gamma) \)
  - various variants
  - aborts execution when condition \( \gamma \) holds
- suspension: \( \text{suspend} \ldots \text{when}(\gamma) \)
  - various variants
  - suspends execution when condition \( \gamma \) holds
- concurrent execution: \( \{ \ldots \} \parallel \{ \ldots \} \)
- \ldots
Causal Dependencies

module P2(bool o)
{
    bool x;
    o = x;
    x = true;
}

- value for o holds for the whole step
- both actions are executed according to their data dependencies
- P2 is causally correct in sense of Quartz

module P3(bool o)
{
    if(!o)
        pause;
    o = true;
}

- if o=true is reached depends on o
- o = true would lead to a valid execution of the program
- P3 is not causally correct in sense of Quartz
Outline

1. Introduction
2. Preliminaries
   - Quartz
   - Hoare Calculus
3. A Hoare Calculus for Quartz
4. Conclusion
Why Hoare...

**model checking**
- fully automatic
- suffers from state-space explosion problem
- enumerates all possible values

**interactive verification based on Hoare calculus**
- interactive (semi-automatic)
- requires additional invariants
- allows abstraction from the size of data structures as well as the data-types itself

An integration of model checking and interactive verification is desired.
Hoare Calculus

nothing:
\[ \{ \Phi \} \text{nothing} \{ \Phi \} \]

assign:
\[ \{ [\Phi]_x^{\tau} \} x = \tau \{ \Phi \} \]

sequence:
\[ \{ \Phi_1 \} S_1 \{ \Phi_2 \} \quad \{ \Phi_2 \} S_2 \{ \Phi_3 \} \]
\[ \{ \Phi_1 \} S_1 ; S_2 \{ \Phi_3 \} \]

conditional:
\[ \{ \sigma \land \Phi \} S_1 \{ \Psi \} \quad \{ \neg \sigma \land \Phi \} S_2 \{ \Psi \} \]
\[ \{ \Phi \} \text{if}(\sigma) \ S_1 \ \text{else} \ S_2 \{ \Psi \} \]

loop:
\[ \{ \sigma \land \Phi \} S \{ \Phi \} \]
\[ \{ \Phi \} \text{while}(\sigma) \ S \{ \neg \sigma \land \Phi \} \]

weaken:
\[ \models \Phi_1 \rightarrow \Phi_2 \quad \{ \Phi_2 \} S \{ \Phi_3 \} \quad \models \Phi_3 \rightarrow \Phi_4 \]
\[ \{ \Phi_1 \} S \{ \Phi_4 \} \]
Hoare Calculus

nothing:

\[
\{ \Phi \} \text{nothing} \{ \Phi \}
\]

assign:

\[
\{ [\Phi]^\tau_x \} x = \tau \{ \Phi \}
\]

sequence:

\[
\{ \Phi_1 \} S_1 \{ \Phi_2 \} \quad \{ \Phi_2 \} S_2 \{ \Phi_3 \}
\]

\[
\{ \Phi_1 \} S_1; S_2 \{ \Phi_3 \}
\]

conditional:

\[
\{ \sigma \land \Phi \} S_1 \{ \Psi \} \quad \{ \neg \sigma \land \Phi \} S_2 \{ \Psi \}
\]

\[
\{ \Phi \} \text{if}(\sigma) S_1 \text{else} S_2 \{ \Psi \}
\]

loop:

\[
\{ \sigma \land \Phi \} S \{ \Phi \}
\]

\[
\{ \Phi \} \text{while}(\sigma) S \{ \neg \sigma \land \Phi \}
\]

weaken:

\[
\models \Phi_1 \rightarrow \Phi_2 \quad \{ \Phi_2 \} S \{ \Phi_3 \} \quad \models \Phi_3 \rightarrow \Phi_4
\]

\[
\{ \Phi_1 \} S \{ \Phi_4 \}
\]
A Hoare calculus for Quartz

- defining a Hoare calculus only requires the definition of a Hoare rule for each statement
- it is possible to synthesis a Quartz program to sequential code and then apply the classical Hoare calculus
Outline

1 Introduction

2 Preliminaries
   - Quartz
   - Hoare Calculus

3 A Hoare Calculus for Quartz

4 Conclusion
problems defining a Hoare calculus for Quartz on statement level

- inputs are read in each macro step
  ⇒ reaching a **pause**: update inputs and depended conditions
- each statement rule requires to regard many cases
- macro step must be identified
- all variable updates have to be done synchronously
- in case no assignment is done the default value must be used
Problems Defining a Hoare Calculus - Many Cases

\{ \Phi \} \ S_1; S_2 \ {\Psi} \Rightarrow

- case: \ inst(S_1) \land \ inst(S_2)
- case: \ \neg inst(S_1) \land \ inst(S_2)
- case: \ inst(S_1) \land \ \neg inst(S_2)
- case: \ \neg inst(S_1) \land \ \neg inst(S_2)

even worse: \ \{ \Phi \} \ S_1 \parallel S_2 \ {\Psi}
Problems Defining a Hoare Calculus for Quartz

problems defining a Hoare calculus for Quartz on statement level

- inputs are read in each macro step
  - ⇒ reaching a **pause**: update inputs and depended conditions
- each statement rule requires to regard many cases
- macro step must be identified
- all variable updates have to be done synchronously
- in case no assignment is done the default value must be used
Macro Step Behaviour

Each variable in a macro step has a unique value. Either determined by

- an delayed assignment in the previous step,
- an immediate assignment in the current step or
- a type dependent default value
Macro Step Behaviour

P4

```c
if (a) {
    x = 5;
    y = true;
} else {
    y = false;
}
pause;
```

P5

```c
if (a) {
    x = 5;
    y = true;
} else {
    y = false;
}
if (!y) x = 3;
pause;
```

P6

```c
if (a) {
    x = 5;
    y = true;
} else {
    y = false;
}
if (!y & b) x = 3;
pause;
```
Macro Step Behaviour

P4
if (a) {
  x = 5;
  y = true;
} else {
  y = false;
}
pause;

!a ⇒ x = 0

P5
if (a) {
  x = 5;
  y = true;
} else {
  y = false;
}
if (!y) x = 3;
pause;

P6
if (a) {
  x = 5;
  y = true;
} else {
  y = false;
}
if (!y & b) x = 3;
pause;
Macro Step Behaviour

P4
\[
\text{if (a) \{ no default value required} \\
\text{\hspace{1cm} x = 5; } \\
\text{\hspace{1cm} y = true; } \\
\text{\hspace{1cm} } \text{else \{ } \\
\text{\hspace{2cm} y = false; } \\
\text{\hspace{1cm} } \text{\}} \\
\text{\hspace{1cm} pause; } \\
\text{!a} \Rightarrow x = 0
\]

P5
\[
\text{if (a) \{ no default value required} \\
\text{\hspace{1cm} x = 5; } \\
\text{\hspace{1cm} y = true; } \\
\text{\hspace{1cm} } \text{else \{ } \\
\text{\hspace{2cm} y = false; } \\
\text{\hspace{1cm} } \text{\}} \\
\text{\hspace{1cm} if (!y) x = 3; } \\
\text{\hspace{1cm} pause; } \\
\text{!a} \Rightarrow !y
\]

P6
\[
\text{if (a) \{ no default value required} \\
\text{\hspace{1cm} x = 5; } \\
\text{\hspace{1cm} y = true; } \\
\text{\hspace{1cm} } \text{else \{ } \\
\text{\hspace{2cm} y = false; } \\
\text{\hspace{1cm} } \text{\}} \\
\text{\hspace{1cm} if (!y & b) x = 3; } \\
\text{\hspace{1cm} pause; } \\
\text{!a} \Rightarrow !y
\]
Macro Step Behaviour

P4

```java
if (a) {
    x = 5;
    y = true;
} else {
    y = false;
}
pause;

!a\Rightarrow x = 0
```

P5

```java
if (a) {
    x = 5;
    y = true;
} else {
    y = false;
}
if (!y) x = 3;
pause;

no default value required
```

P6

```java
if (a) {
    x=5;
    y = true;
} else {
    y = false;
}
if (!y & b) x = 3;
pause;

!a \& \!b \Rightarrow x = 0
```

!a\Rightarrow !y
Hoare Calculus

if (a) {
    ...
    pause;
} else {
    x = 3;
}

if (x >= 2)
    ...

if (a) x = 7;
pause;

- variable’s default value may be read!
- determine the necessary of the default value cannot be done locally (on statement level)
Hoare Calculus

if (a) {
  ...
  pause;
} else {
  x = 3;
}

if (x >= 2)
  ...

if (a) x = 7;
pause;

- variable’s default value may be read!
- determine the necessary of the default value cannot be done locally (on statement level)
if (a) {
    ...
    pause;
} else {
    x = 3;
}
if (x >= 2)
    ...
if (a) x = 7;
pause;

- variable’s default value may be read!
- determine the necessary of the default value cannot be done locally (on statement level)
variable’s default value may be read!

determine the necessary of the default value cannot be done locally (on statement level)
Definition of a Hoare Calculus for Quartz

- define two-stage Hoare-like rules.
  1. identify macro step
  2. reason about the macro step
Definition of a Hoare Calculus for Quartz

- define two-stage Hoare-like rules.
  1. identify macro step
  2. reason about the macro step

- split the verification process into these stages.
  1. source-code transformation that collects all macro step’s actions
  2. reason about code in a certain normal form
Synchronous Tuple Assignments (STA)

collecting all actions in synchronous tuple assignments (STAs)

Definition (Synchronous Tuple Assignment (STA))

Given that $x_1 = \tau_1, \ldots, x_m = \tau_m$ and next$(y_1) = \pi_1, \ldots,$ next$(y_m) = \pi_m$ are assignments with pairwise different left-hand side expressions $x_i$ and $y_i$, and given that these assignments are causally ordered such that there are no read-after-write conflicts, i.e. that $\tau_i$ only has occurrences of $x_1, \ldots, x_{i-1}$, then we call the following statement a synchronous tuple assignment:

$$(x_1, \ldots, x_m) . (y_1, \ldots, y_n) = (\tau_1, \ldots, \tau_m) . (\pi_1, \ldots, \pi_n)$$
Definition (Quartz Programs in STA Form)

A Quartz program is in synchronous tuple assignment (STA) form if all its actions are STAs and between the execution of two STAs at least one `pause` is executed.
Fibonacci Numbers

module Fib(nat ?i,f,event !r)
{
  nat k,g,n;
  n = i;
  if(n <= 0)
    f=0;
  else {
    k = 1;
    g = 0;
    f = 1;
    while(k != n) {
      next(g) = f;
      next(f) = f+g;
      next(k) = k+1;
      l: pause;
    }
  }
  emit(r);
}
module Fib(nat ?i,f,event !r)
{
    nat k,g,n;
    n = i;
    if(n <= 0)
        f=0;
    else {
        k = 1;
        g = 0;
        f = 1;
        while(k != n) {
            next(g) = f;
            next(f) = f+g;
            next(k) = k+1;
        l: pause;
        }
    }
    emit(r);
}
 module FSA(nat ?i,f,event r)
{
    nat k,g,n,l;
    do 
    {
        case 
            (l == 0) do /*State 0*/
                (n,r,k,g,f).(g,f,k,l) =
                    (i,n <= 0,1,0,(n > 0?1:0)).
                (f,f+g,k+1,(n > 0& n! = k?1:2));
            (l == 1) do /*State 1*/
                (r).(g,f,k,l) =
                    (n == k).
                (f,f+g,k+1,(n! = k?1:2));
        default
            nothing;
        pause;
    } while (l != 2);
}
module Fib(nat ?i,f,event !r)
{
    nat k,g,n;
    n = i;
    if(n <= 0)
    f = 0;
    else {
        k = 1;
        g = 0;
        f = 1;
        while(k != n) {
            next(g) = f;
            next(f) = f+g;
            next(k) = k +1;
            l: pause;
        }
    }
    emit(r);
}

module FSH(nat ?i,f,event !r)
{
    nat k,g,n;
    if(n <= 0) {
        (n,f,r).() = (i,0, true).();
    } else {
        (n,k,g,f,r).(g,f,k) =
        (i,1,0,1,k==n).(f,f+g,k+1);
        while(k!=n) {
            pause;
            (r).(g,f,k) = (k==n).
            (f,f+g,k+1);
            }
    }
}
module FSH(nat ?i,f,event !r) {
  nat k,g,n;
  if(n<=0) {
    (n,f,r).() = (i,0,true).();
  } else {
    (n,k,g,f,r).(g,f,k) =
    (i,1,0,1,k==n).(f,f+g,k+1);
    while(k!=n) {
      pause;
      (r).(g,f,k) = (k==n).
      (f,f+g,k+1);
    }
  }
}

- structure is preserved
- assignment are shifted and/or duplicated
- same invariants are usable
Required Rules

STA Rule

\[
\{ \left[ \cdots \left[ \Phi \pi_1^{\tau_1}, \ldots, \pi_n^{\tau_n} \right]^{x_n} \cdots \right]^{x_1} \} (x_1, \ldots, x_m) \cdot (y_1, \ldots, y_n) = (\tau_1, \ldots, \tau_m) \cdot (\pi_1, \ldots, \pi_n) \{ \Phi \}\]

Pause Rule

\[
\{ \left[ \cdots \Phi \cdots \right]^{i_1, \ldots, i_n} y_1^{y_1} \cdots y_n^{y_n} \} \text{ pause } \{ \Phi \}\]
STA form is reasonable

- all Quartz programs representable
- Hoare rules are easily adaptable
- code structure is preservable
Outline

1. Introduction

2. Preliminaries
   - Quartz
   - Hoare Calculus

3. A Hoare Calculus for Quartz

4. Conclusion
A Hoare Calculus for Quartz

Summary

- discussed issues defining Hoare rules for Quartz
- introduced a new kind of normal form for Quartz
- showed the practicability of STA form
- extended the set of Hoare rules for Quartz
What needs to be done?

- further work: defining a structure preserving transformation (partially done)
- alternative idea: defining rules on AIF level, but user provides invariants and chose rules on source-code level. Advantages are:
  - reuse of schizophrenia and causality techniques
  - AIF transformations are apply-able before verification
  - no need for STA transformation (implicitly usage)
  - invariants of source code are usable
  - no need to verify the compiling procedure

thank you for your attention